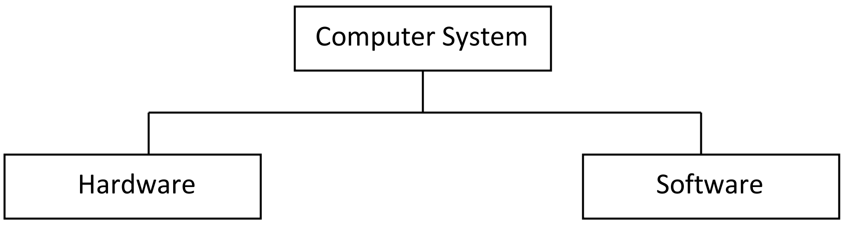
UNIT-I

Computer Systems:

A computer is a system made of two major components. Those are

- Hardware
- Software



Computer Hardware:

The hardware component of the computer system consists of five parts: input devices, central processing unit (CPU), primary storage, output devices, and auxiliary storage devices.

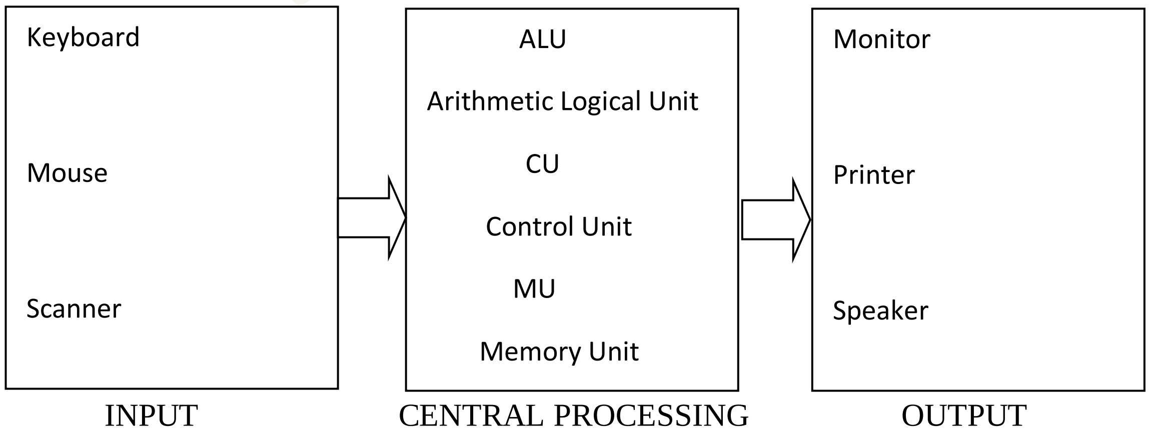
The **input device** is usually a keyboard where programs and data are entered into the computer.

Ex: Mouse, Keyboard, touch screen.

The **central processing unit (CPU)** is responsible for executing instructions such as arithmetic calculations, comparisons among the data, and movement of the data inside the system.

The hardware of a computer system can be classified as

1) Input Devices(I/P)
2) Processing Devices (CPU)
3) Output Devices(O/P)



ALU: It performs the Arithmetic and Logical Operations such as +,-,*,/ (Arithmetic Operators) &&, || (Logical Operators)
CU: Every Operation such as storing, computing and retrieving the data should be governed by the control unit.

MU: The Memory unit is used for storing the data.

The Memory unit is classified into two types.
They are 1) Primary Memory
2) Secondary Memory

Primary Memory: The following are the types of memories which are treated as primary

a) **ROM:** It represents Read Only Memory that stores data and instructions even when the computer is turned off. The Contents in the ROM can't be modified once if they are written. It is used to store the BIOS information.
b) **RAM:** It represents Random Access Memory that stores data and instructions when the computer is turned on. The contents in the RAM can be modified any no. of times by instructions. It is used to store the programs under execution.
c) **CACHE MEMORY:** It is used to store the data and instructions referred by processor.

Secondary Memory: The following are the different kinds of memories

a) **Magnetic Storage:** The Magnetic Storage devices store information that can be read, erased and rewritten a number of times.
Example: Floppy Disks, Hard Disks, Magnetic Tapes
b) **Optical Storage:** The optical storage devices that use laser beams to read and write stored data.
Example: CD (Compact Disk)
DVD (Digital Versatile Disk)

Computer Software:

Software of a computer system can be referred as anything which we can feel and see.
Example: Windows, icons
Computer software is divided in to two broad categories:

- System Software
- Application Software

System software: System software manages the computer resources .It provides the interface between the hardware and the users.
Application software: Application software is directly responsible for helping users to solve their problems.

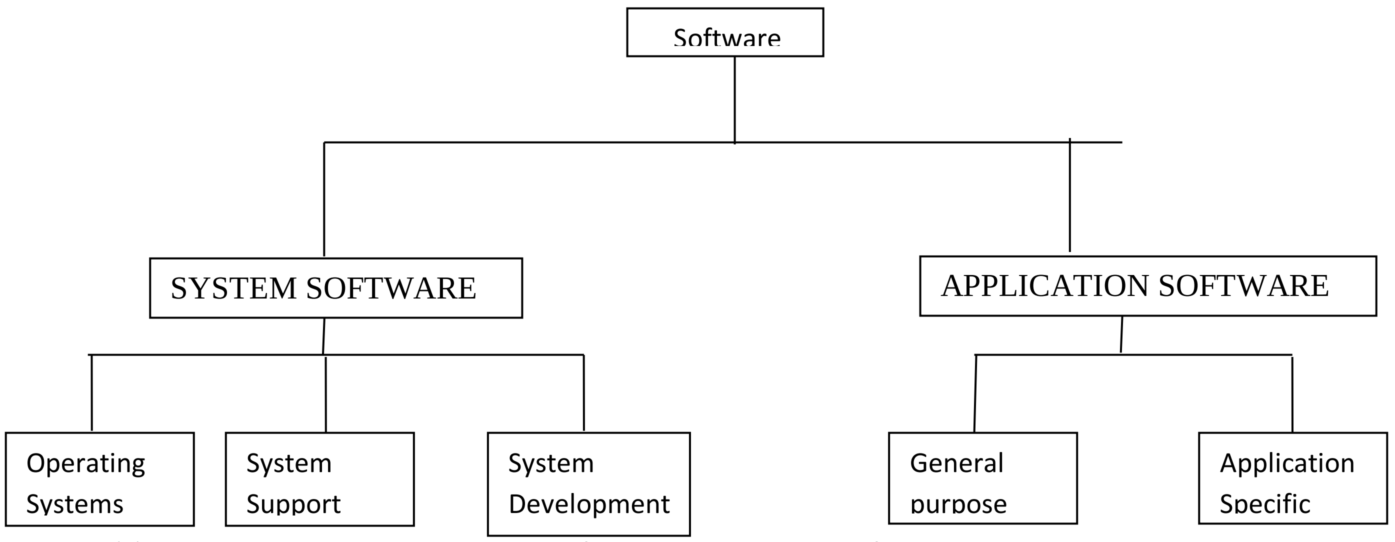


Fig: Types of software

System Software:

System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The **operating system** provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consists of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category, **system development software**, includes the language translators that convert programs into machine language for execution, debugging tools to ensure that the programs are error free and computer –assisted software engineering (CASE) systems.

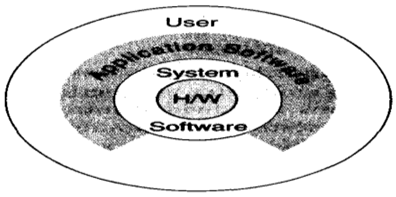
Application software:

Application software is broken in to two classes: general-purpose software and application –specific software. **General purpose software** is purchased from a software developer and can be used for more than one application. Examples of general purpose software include word processors, database management systems, and computer aided design systems. They are labelled general purpose because they can solve a variety of user computing problems.

Application –specific software can be used only for its intended purpose.

A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed they cannot be used for other generalized tasks.

The relationship between system and application software is shown below. In this figure, each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hard ware. The opening at the bottom of the figure is the path followed by the user who interacts directly with the operating system when necessary.



Relationship Between System and Application Software

COMPUTER LANGUAGES:

To write a program (tells what to do) for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages. The following is the summary of computer languages

1940's	--	Machine Languages
1950's	--	Symbolic Languages
1960's	--	High Level Languages

Machine Language:

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language which is made of streams of 0's and 1's. The instructions in machine language must be in streams of 0's and 1's. This is also referred as binary digits. These are so named as the machine can directly understood the programs

Advantages:

1) High speed execution
2) The computer can understood instructions immediately
3) No translation is needed.

Disadvantages:

1) Machine dependent
2) Programming is very difficult
3) Difficult to understand
4) Difficult to write bug free programs
5) Difficult to isolate an error

Example Addition of two numbers

2 → 0 0 1 0

+ 3 → 0 0 1 1

---		---------------
5	←	0 1 0 1
---		---------------

Symbolic Languages (or) Assembly Language:

In the early 1950's Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language. These early programming languages simply mirrored the machine languages using symbols or mnemonics to represent the various language instructions. These languages were known as symbolic languages. Because a computer does not understand symbolic language it must be translated into the machine language. A special program called an **Assembler** translates symbolic code into the machine language. Hence they are called as Assembly language.

Advantages:

1) Easy to understand and use
2) Easy to modify and isolate error
3) High efficiency
4) More control on hardware

Disadvantages:

1) Machine Dependent Language
2) Requires translator
3) Difficult to learn and write programs
4) Slow development time
5) Less efficient

Example:

	2	PUSH 2,A
	3	PUSH 3,B
+		ADD A,B
	5	PRINT C

HIGH-LEVEL Languages:

The symbolic languages greatly improved programming efficiency they still required programmers to concentrate on the hardware that they were using working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problems being solved led to the development of high-level languages.

High-level languages are portable to many different computers allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer.

C	A systems implementation Language
C++	C with object oriented enhancements
JAVA	Object oriented language for internet and general applications using basic C syntax

Advantages:

1) Easy to write and understand
2) Easy to isolate an error
3) Machine independent language
4) Easy to maintain
5) Better readability
6) Low Development cost
7) Easier to document
8) Portable

Disadvantages:

1) Needs translator
2) Requires high execution time
3) Poor control on hardware
4) Less efficient

Example: C language

```
#include<stdio.h>
void main()
{
        int a,b,c;

        scanf("%d%d%",&a,&b);
        c=a+b;
        printf("%d",c);
}
```

Difference between Machine, Assembly, High Level Languages

Feature	**Machine**	**Assembly**	**High Level**
Form	0's and 1's	Mnemonic codes	Normal English
Machine Dependent	Dependent	Dependent	Independent
Translator	Not Needed	Needed(Assembler)	Needed(Compiler)
Execution Time	Less	Less	High
Languages	Only one	Different Manufacturers	Different Languages
Nature	Difficult	Difficult	Easy
Memory Space	Less	Less	More

Language Translators

These are the programs which are used for converting the programs in one language into machine language instructions, so that they can be executed by the computer.

1) **Compiler:** It is a program which is used to convert the high level language programs into machine language
2) **Assembler:** It is a program which is used to convert the assembly level language programs into machine language
3) **Interpreter:** It is a program; it takes one statement of a high level language program, translates it into machine language instruction and then immediately executes the resulting machine language instruction and so on.

Comparison between a Compiler and Interpreter

COMPILER	**INTERPRETER**
A Compiler is used to compile an entire program and an executable program is generated through the object program	An interpreter is used to translate each line of the program code immediately as it is entered
The executable program is stored in a disk for future use or to run it in another computer	The executable program is generated in RAM and the interpreter is required for each run of the program
The compiled programs run faster	The Interpreted programs run slower
Most of the Languages use compiler	A very few languages use interpreters.

Algorithm

Definition:- A step by step specification of a method to solve the given engineering problem.

Examples

1. **Write an algorithm to add two numbers entered by user.**

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
         sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

2. **Write an algorithm to find the largest among three different numbers entered by user.**

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
               If a>c
                  Display a is the largest number.
               Else
                  Display c is the largest number.
        Else
            If b>c
                Display b is the largest number.
```

```
                    Else
                        Display c is the greatest number.
        Step 5: Stop
```

3. **Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.**

```
Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant
         D←b2-4ac
Step 4: If D≥0
              r1←(-b+√D)/2a
              r2←(-b-√D)/2a
              Display r1 and r2 as roots.
        Else
              Calculate real part and imaginary part
              rp←b/2a
              ip←√(-D)/2a
              Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop
```

4. **Write an algorithm to find the factorial of a number entered by user.**

```
Step 1: Start
Step 2: Declare variables n,factorial and i.
Step 3: Initialize variables
          factorial←1
          i←1
Step 4: Read value of n
Step 5: Repeat the steps until i=n
     5.1: factorial←factorial*i
     5.2: i←i+1
Step 6: Display factorial
Step 7: Stop
```

5. **Write an algorithm to check whether a number entered by user is prime or not.**

```
Step 1: Start
Step 2: Declare variables n,i,flag.
Step 3: Initialize variables
        flag←1
        i←2
Step 4: Read n from user.
Step 5: Repeat the steps until i<(n/2)
     5.1 If remainder of n÷i equals 0
            flag←0
            Go to step 6
     5.2 i←i+1
Step 6: If flag=0
           Display n is not prime
        else
           Display n is prime
Step 7: Stop
```

6. **Write an algorithm to find the Fibonacci series till term≤1000.**

```
Step 1: Start
Step 2: Declare variables first_term,second_term and temp.
Step 3: Initialize variables first_term←0 second_term←1
Step 4: Display first_term and second_term
Step 5: Repeat the steps until second_term≤1000
     5.1: temp←second_term
     5.2: second_term←second_term+first term
     5.3: first_term←temp
     5.4: Display second_term
Step 6: Stop
```

Properties of Algorithm

- ✓ **Input:** An Algorithm has zero or more inputs, taken from a specified set of objects.
- ✓ **Output:** An algorithm has one or more outputs, which have a specified relation to the inputs.
- ✓ **Effectiveness:** All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.
- ✓ **Definiteness:** Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- ✓ **Finiteness:** The algorithm must always terminate after a finite number of steps.

Methods of Expressing an Algorithm

1. Using English Language
2. Using a Flow-Chart
3. Using a Pseudo code

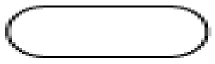
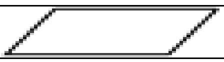
1. **Using English Language:** In this method to express an algorithm we use general engish language.

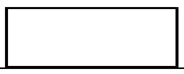
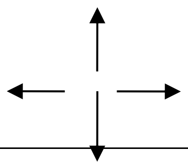
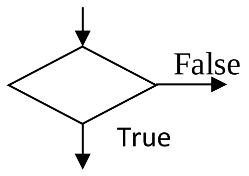
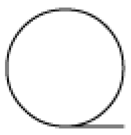
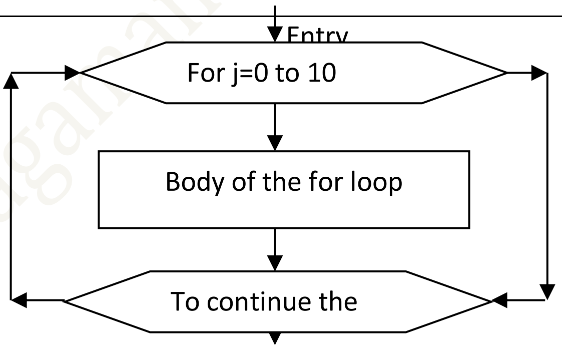
   Example: Write an algorithm to find the largest among three different numbers entered by user.

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
               If a>c
                  Display a is the largest number.
               Else
                  Display c is the largest number.
          Else
              If b>c
                 Display b is the largest number.
              Else
                 Display c is the greatest number.
Step 5: Stop
```

2. **Using a Flow-Chart:** The graphical or visual or pictorial representation of an algorithm is called flow-chart. In this method we use some special symbols to express an algorithm.

   Standard Flow Chart Symbols

1. Terminal (Start or Stop symbol)	
2. Input/output	

3. Processing	
4. **Flow Lines**: These are arrow mark symbols used to connect two boxes and to indicate the direction of data or processing flow.	
5. **Decision Box:** This is a diamond shaped box, which is used to indicate logical checking and gives decision after comparing between two or more objects.(Eg. True or False; Yes or No)	
6. **Connecter:** This is a Circular-shaped symbol used to connect different parts of flowchart. When the flow chart is lengthy, i is split into different pages. Then these connectors are used to connect between these pages at the beginning and at the end of each page.	
7. **Loop Symbol**: this symbol looks like a hexagon. This symbol is used for implementation of loops only. Four flow lines are associated with this symbol. Two lines are used to indicate the sequence of the program and remaining two are used to show the looping area, that is, from beginning to the end.	 Entry For j=0 to 10 Body of the for loop To continue the

Flow Chart Examples:

I. **Algorithm and Flow-Chart to read the name and print the name**.

<u>Algorithm</u>

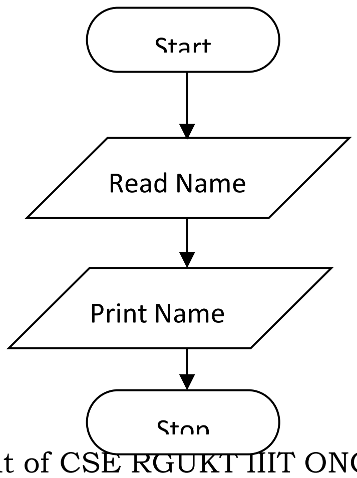
Step-1: Start

Step-2: Read input name

Step-3: Print the name

Step-4: Stop

<u>Flow-Chart</u>



II. Algorithm and Flow-Chart to add two numbers entered by the user.

Algorithm

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the

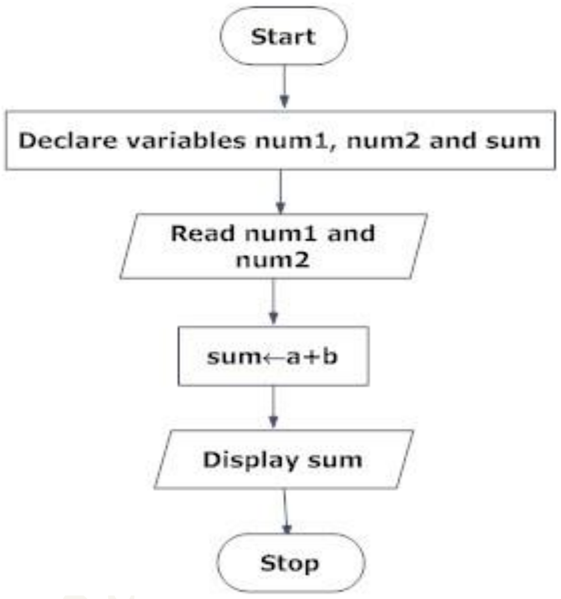
result to sum.

sum←num1+num2

Step 5: Display sum

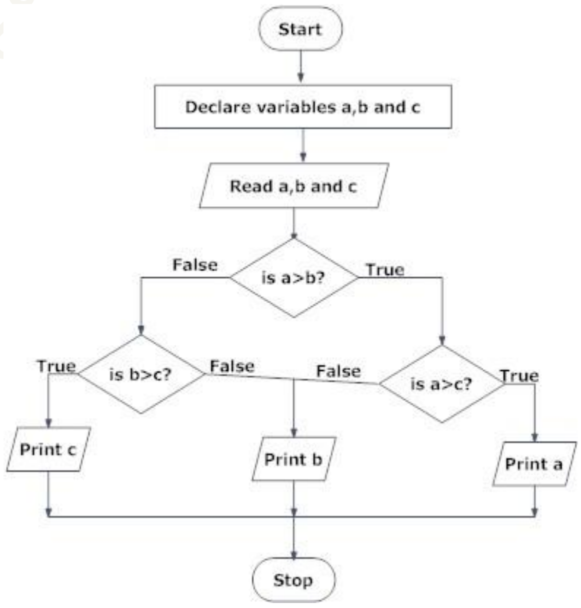
Step 6: Stop

Flow-Chart



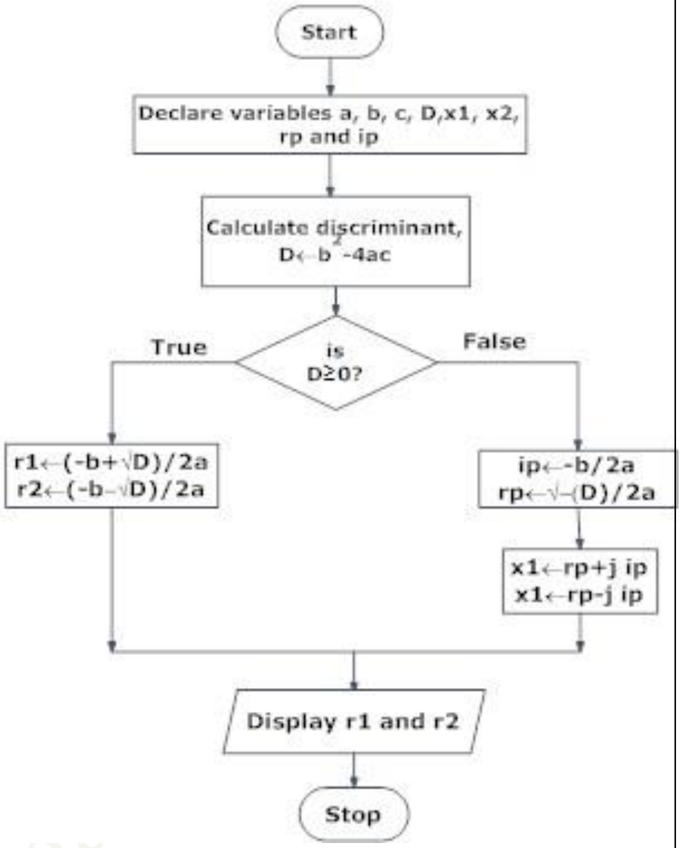
III. Algorithm and Flow-Chart to find the largest among three different numbers entered by user.

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
           If a>c
              Display a is the largest number.
           Else
              Display c is the largest number.
        Else
           If b>c
              Display b is the largest number.
           Else
             Display c is the greatest number.
Step 5: Stop
```



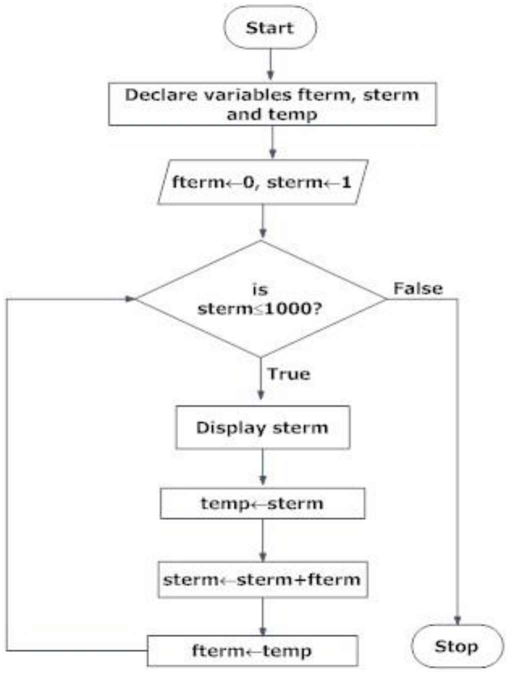
IV. Algorithm & Flow-Chart to find all roots of a quadratic equation $ax^2+bx+c=0$.

```
Step 1: Start
Step 2: Declare variables a, b, c, D, x1,
         x2, rp and ip;
Step 3: Calculate discriminant
           D←b2-4ac
Step 4: If D≥0
            r1←(-b+√D)/2a
            r2←(-b-√D)/2a
            Display r1 and r2 as roots.
        Else
            Calculate real part and
             imaginary part
            rp←b/2a
            ip←√(-D)/2a
            Display rp+j(ip) and rp-j(ip)
         as roots
Step 5: Stop
```



V. Algorithm & Flow-Chart to find the Fibonacci series till term≤1000.

```
Step 1: Start
Step 2: Declare variables first_term,second_term
        and temp.
Step 3: Initialize variables first_term←0
        second_term←1
Step 4: Display first_term and second_term
Step 5: Repeat the steps until second_term≤1000
       5.1: temp←second_term
       5.2: second_term←second_term+first term
       5.3: first_term←temp
       5.4: Display second_term
Step 6: Stop
```



3. **Using a Pseudocode**

   Definition:- Pseudocode is a simple way of writing programming code in English. Pseudocode is not actual programming language. It uses short phrases to write code for programs before you actually create it in a specific language. Once you know what the program is about and how it will function, then you can use pseudocode to create statements to achieve the required results for your program.

Problem Solving:- To solve a given problem we should follow the following steps.

Problem solving steps

- Problem Specification
- Formulation
- Algorithm
- Coding

Examples:

1. GCD of two integer values.

A) Problem Specification: Write a program to find GCD of two integers and print that value.

B) Formulation:

- Let X and Y are integers.
- If X=0 then GCD is Y.
- If Y=0 then GCD is X.
- Method example:

	X	Y
	48	64
Iteration-1:	48	16
Iteration-2:	0	16

C) Algorithm:

```
Step-1:  Start
Step-2:  Read X and Y as input.
Step-3: P=X; Q=Y;
Step-4: Start a loop with condition as P≠0 and Q≠0.
Step-5: In every iteration do the following
        Step-5a: If P≥Q update P as P%Q.
        Step-5b: If Q>P update Q as Q%P.
Step-6: If X=0 update GCD as Y.
    Else if Y=0 update GCD as X.
Step-7: Print GCD.
Step-8:  Stop
```

D) Coding:

```
#include <stdio.h>
void main()
{
int p,q,x,y,gcd_result;
clrscr();
printf("Enter the Integer values for x,y");
scanf("%d%d",&x,&y);
p=x,q=y;
        if(x==0)
        printf("GCD of %d and %d is %d",x,y,y);
        else if(y==0)
        printf("GCD of %d and %d is %d",x,y,x);
        else
        {
                while(x!=0&&y!=0)
                {
                        if(x>=y)
                        {
                                x=x%y;
                        }
                        else
                        {
                                y=y%x;
                        }
                }
                if(x==0)
                gcd_result=y;
                else
                gcd_result=x;
        }
printf("GCD of %d and %d is %d",p,q,gcd_result);
getch();
}
```

2. LEAP YEAR OR NOT

A) Problem Specification: To find the given year is leap year or not

B) Formulation: Every fourth year is a leap year i.e 4,8,12,16...

4 multiple year is leap year except centuryth year

Incase of centuryth year that should be multiple of 400 other wise that is not a leap year

C) Algorithm:

```
Step1: read  year as input
Step2: check if year is centuryth year or not
Step3: if it is centuryth year check year%400==0 then leap year other wise
    not a leap year
Step4: if it is not a centuryth year check year%4==0 then leap year other
    wise not a leap year
```

D) Code:

```
void main()
{
    int year;
    printf("Enter year:");
    scanf("%d",&year);
    if(year%400==0)
        printf("Given year is leap year");
    else if(year%4==0)
```

```
            printf("Given year is leap year");
        else
            printf("Given year is not a leap year");
        }
```

<u>Introduction to C-language:</u>

A Brief History of C

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. **BCPL** and **B** are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

Advantages of C language:

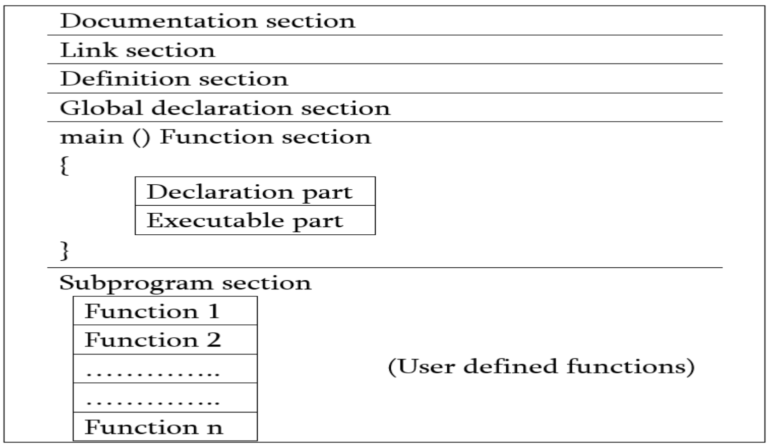
1. C language is a building block for many other currently known languages. C language has variety of data types and powerful operators. Due to this, programs written in C language are efficient, fast and easy to understand.

2. There are only 32 keywords in ANSI C and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

3. Another important advantage of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library this makes us easier to add our own functions to C library. Due to the availability of large number of functions, the programming task becomes simple.

4. C language is a structured programming language. This makes user to think of a problem in terms of function modules or blocks. Collection of these modules makes a complete program. This modular structure makes program debugging, testing and maintenance easier.

Structure of C-Program



Documentation Section:- The documentation section contains a set of comment including the name of the program other necessary details. Comments are ignored by compiler and are used to provide documentation to people who reads that code. Comments are be giving in C programming in two different ways:

1. Single Line Comment
2. Multi Line Comment

```
// This is single line comment

/*
This is
multi line
comment
*/
```

Link Section (or) pre-processor Directives :- The link section consists of header files while contains function prototype of Standard Library functions which can be used in program. Header file also consists of macro declaration.

Example:
```
#include <stdio.h>
```
In the above code, stdio.h is a header file which is included using the preprocessing directive “#include”. Learn more about header files in C programming .

Definition Section:- The definition section defines all symbolic constants. A symbolic constant is a constant value given to a name which can't be changed in program.

Example:
```
#define PI 3.14
```

In above code, the PI is a constant whole value is 3.14

Global Declaration:-

In global declaration section, global variables and user defined functions are declared.

main() Function Section:- The main () function section is the most important section of any C program. The compiler start executing C program from **main()** function. The **main()** function is mandatory in C programming. It has two parts:

Declaration Part - All the variables that are later used in the executable part are declared in this part.

Executable Part - This part contains the statements that are to be executed by the compiler.

```
main()
{
   ... .. ...
   ... .. ...
}
```

Subprogram Section:-

The **subprogram section** contains all the user defined functions. A complete and fully functions C program can be written without use of user-defined function in C programming but the code maybe be redundant and inefficient if user-defined functions are not used for complex programs.

Note: Except for the main () function section, other sections of the C program may be omitted as per the requirement of the program.

Basic C Program to Print Hello World

```
// This is the very simple C program to print Hello Word
// This section is documentation section

#include  // Here stdio.h is header file and this section is link section

int main() {      // main()
   printf("Hello world");
   return 0;
}
```

Output

```
Hello world
```

Input and Output Statements in C Language:

Input Statements: Input functions or streams or statements are used to read different types of data (Such as numeric, string, or character) from the keyboard. You can use more than one type of input functions in the program.

Scanf()

Scanf() is the input function of standard library. In C language, scanf() function can be used without giving reference to standard library but you need to include the stdio.h header file. Scanf() function is used to read the data from the keyboard.

Synatx : scanf("list of format specifiers",& list of variables);

The first part of the scanf() syntax is the list of format specifiers which depends on the dfata type of the variable to be read.Thispart should be enclosed in double quotes.The % is used to tell or instruct compiler that next coming character is format specifier.

The second part of the general syntax of scanf() function is the list of variables. Each variable name is start with a symbol & is known as an address operator which refer to the address of the address of that variablewhich is going to be read.

Example1: scanf("%d%d",&a,&b); Here some of the access specifiers are listed below.

Control Character	**Explanation**
%c	A single character
%d	A decimal integer
%e, %f, %g	A floating-point number
%o	An octal number
%s	A string
%x	A hexadecimal number
%n	An integer equal to the number of characters read so far
%u	An unsigned integer
%[]	A set of characters

Output Statements: Output functions or streams or statements are used to display different types of data (Such as numeric, string, or character) and results on the screen. You can use more than one type of output functions in the program.

printf();

printf() is the output function of standard library. In C language, printf() function can be used without giving reference to standard library but you need to include the stdio.h header file.

Syntax: **printf ("list of format specifiers and escape sequences", list of variables);**

Like scanf() function, the general syntax of printf() function has two parts. First part of the printf () syntax refers to the list of format specifiers which help to display data variables according to their data type and a list of escape sequences to print formatted data. The use of \ instructs compiler that next coming character has a special meaning. The list of escape sequences are given below.

Escape sequence	**Purpose**	**Example**	**Output of example**
\n	New line(vertical space)	Printf("sai\nkumar");	Sai kumar
\t	Tab(Horizontal space)	Printf("sai\tkumar");	Sai kumar
\b	Backspace	Printf("sai\b\bkumar");	skumar
\r	Carriage return	Printf("sai\rkumar");	Kumar
\'	Single quote	Printf("don\'t");	Don't
\"	Double quote	Printf("\"hai\"");	"Hai"

Variables: Variables are named memory locations that have a type, such as integer or character.

- Variables are simply names used to refer to some location in memory – a location that holds a value with which we are working. It may help to think of variables as a place holder for a value.

While dealing with the variables in C language we mainly think about 2 things. Those are

1. Variable Declaration.
2. Variable Initialization.

1. **Variable Declaration:** Each variable in our program must be declared and defined. In C, a **declaration** is used to name an object, such as a variable. **Definitions** are used to create the object. With one exception, a variable is declared and defined at the same time.

   A variable declaration specifies a data type and contains a list of one or more variables of that type as follows −

   Syntax: `type variable-list;`

   Here, **type** must be a valid C data type including char, int , float, double, or any user-defined object;

   Variable-list may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float f, salary;
double d;
```

   The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

2. **Variable Initialization:** C variables can be initialized with the help of assignment operator '='.

   Syntax: data_type variable_name=constant/literal/expression;
   or
   variable_name=constant/literal/expression;

   Examples: int a=10; int a=b+c; a=10; a=b+c;

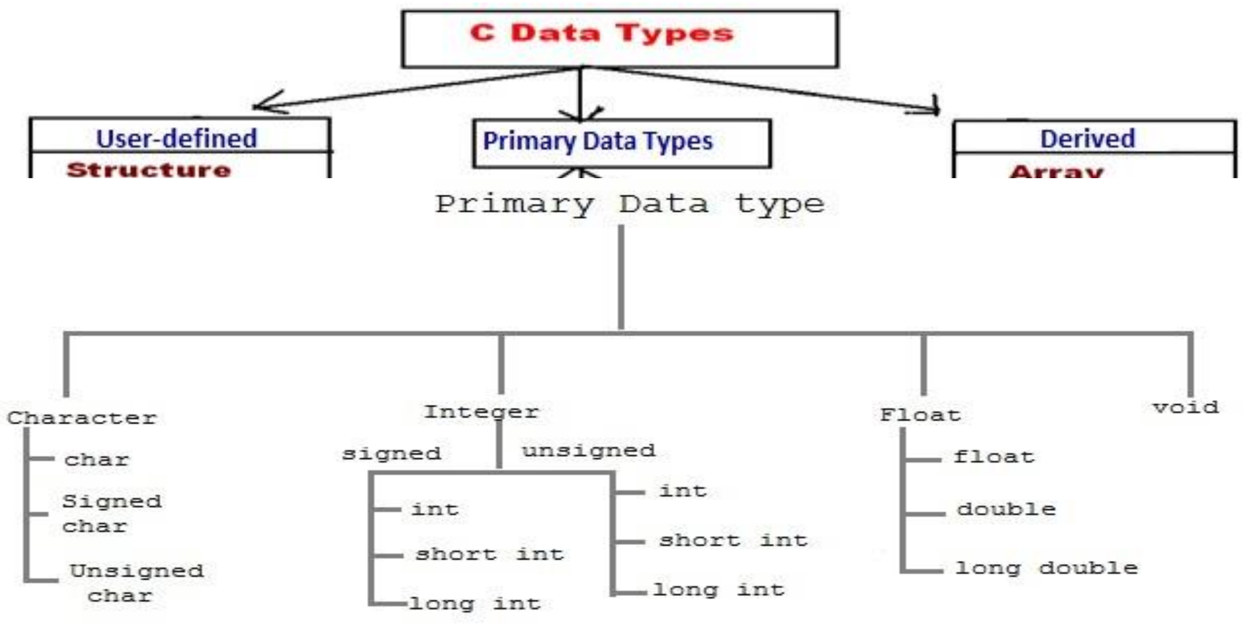
Multiple variables can be initialized in a single statement by single value, for example, a=b=c=d=10;
NOTE: C variables must be declared before they are used in the c program. Also, since C is a case sensitive programming language, therefore the c variables, abc, Abc and ABC are all different.

Data Types:

- **<u>Definition:</u>** Data Type defines the set of values and a set of operations that can be applied on those values.

  The C language has defined a set of Data Types that can be divided into 3 general categories. Those are

  a) Primary Data Types
  b) User-defined Data Types
  c) Derived Data Types



Primary Data Types: These are fundamental data types in C namely integer (**int**), floating (**float**), character (**char**) and **void**.

Integer Data Type

- Integers are whole numbers with a range of values, the range limits being machine dependent.
- Denoted by the keyword int.
- Generally an integer occupies 2 bytes memory space and its value range is limited to -32768 to +32767 (that is, -215 to +215-1).
- A signed integer uses one bit for storing the sign and 15 bits for the number.

To control the range of numbers and storage space, C has three classes of integer storage:

1. short int
2. int
3. long int

A short int requires half the amount of storage than normal integer. The long integers are used to declare a longer range of values and it occupies 4 bytes of storage space.

All three data types have signed and unsigned forms.

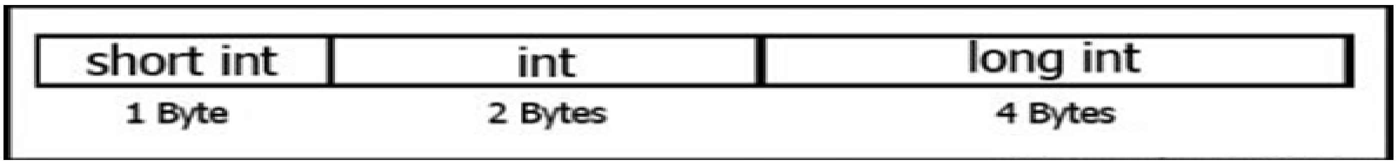
- Unlike signed integers, unsigned integers are always positive and use all the bits for the magnitude of the number. Therefore the range of an unsigned integer will be from 0 to 65535.

Syntax: int <variable name>; like
int num1;
short int num2;
long int num3;

Example: 5, 6, 100, 2500.

Integer Data Type Memory Allocation

Floating point Data Type

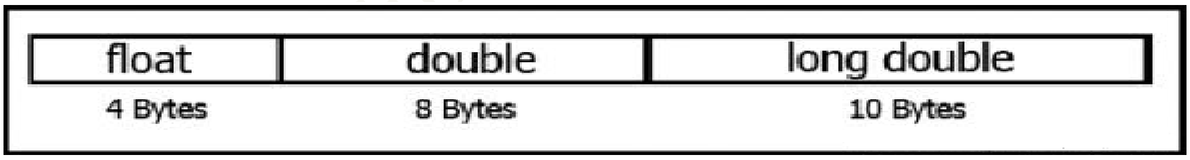


- The **float** data type is used to store fractional numbers (real numbers) within 6 digits of precision.
- Floating point numbers are denoted by the keyword **float**.
- When the accuracy of the floating point number is insufficient, we can use the specifier "**double**" to define the number. "**Double**" is the same as "**float**" but has double the precision and takes up twice as much memory space (8 bytes) as float.
- For even more precision we can use "long double" which occupies 10 bytes of memory space.

Syntax: float <variable name>; like:
float num1;
double num2;
long double num3;

Example: 9.125, 3.1254.

Floating Point Data Type Memory Allocation



Character Data Type

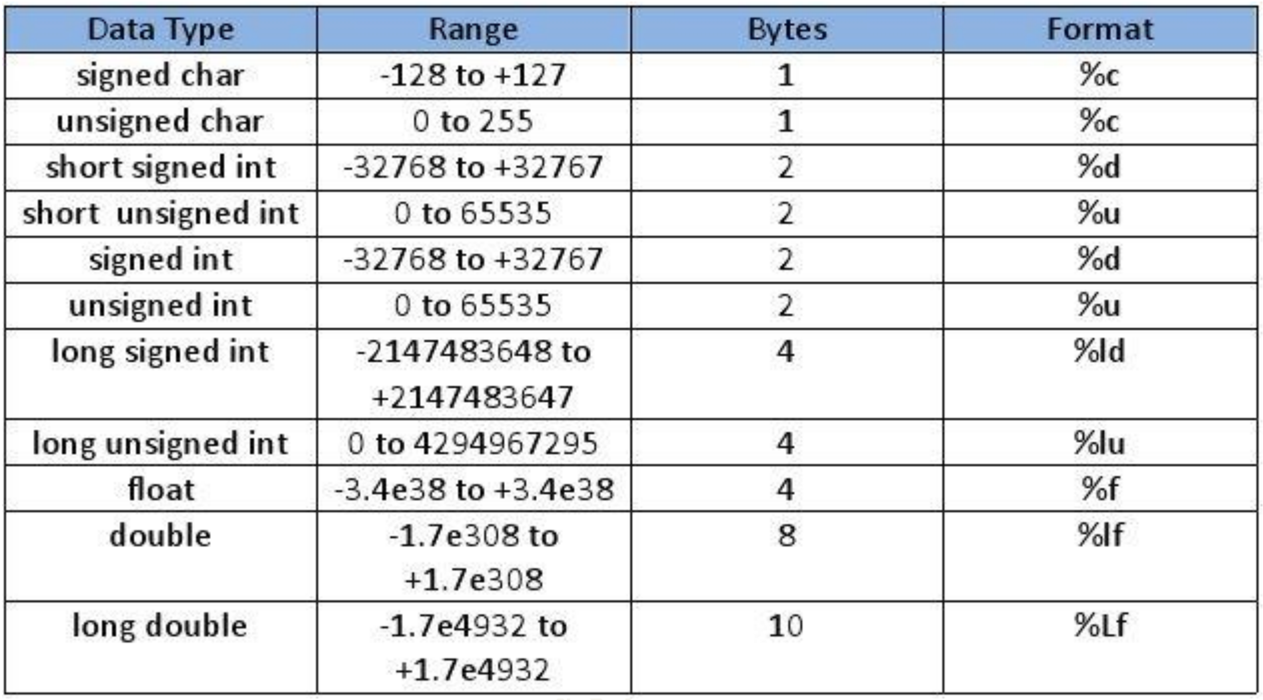
- The character type variable can hold a single character.
- "Char," like "int," may be singed or unsigned. Both signed and unsigned occupy 1 byte each but have different ranges. Unsigned characters have values between 0 and 255; signed characters have values from –128 to 127.
- Denoted by **char**.

Syntax: char <variable name>; like:
char ch = 'a';

Example: a, b, g, S, j.

Void Data Type

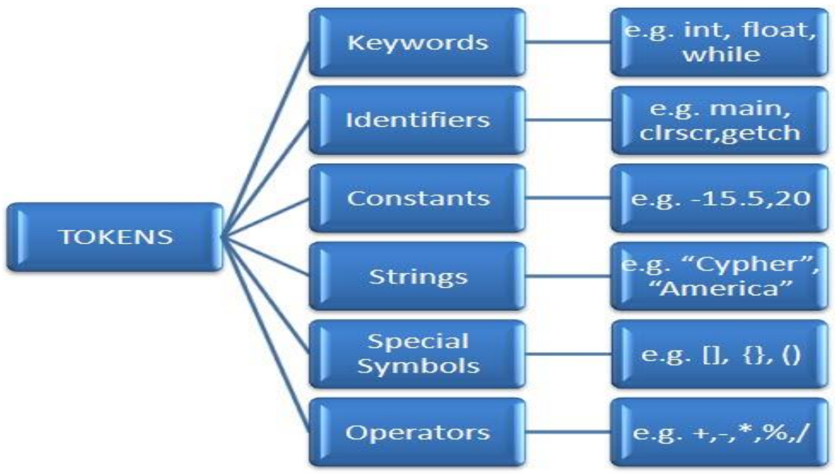
- The “void” data type returns no data; it is not a variable like the other types. It may be used with functions that need to be run without returning any values.
- For example, we can declare “main()” as a void type because it does not return any value.



Data Type	Range	Bytes	Format
signed char	-128 to +127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf

C Tokens:

- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual units in a C program are known as C tokens.
- C tokens are of six types. They are,



Keywords

➢ **Definition:** Keywords are preserved words that have special meaning in C language. The meaning has already been described. These meaning cannot be changed. There are total 32 keywords in C language.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
const	extern	return	union
char	float	short	unsigned
continue	for	signed	volatile
default	goto	sizeof	void
do	if	static	while

Identifiers:

➢ **Definition:-** An **identifier** is a string of alphanumeric characters that begins with an alphabetic character or an underscore character that are used to represent various **programming** elements such as variables, functions, arrays, structures, unions and so on. Actually, an **identifier** is a user-**defined** word.

➢ **Rules to Define an Identifier:-**

1. An Identifier can only have alphanumeric characters (a-z, A-Z, 0-9) and underscore (_).
2. The first character of an identifier can only contain alphabet (a-z , A-Z) or underscore (_).
3. Identifiers are also case sensitive in C. For example *name* and *Name* are two different identifier in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

Examples: **Valid Identifiers** : a, student_name, _name.
Invalid Identifiers : $sum, 2names, sum-sal, int, student name.

Constants:

➢ **Definition: -** Constant in C means the content whose value does not change at the time of execution of a program.

➢ A constant is a value or an identifier whose value cannot be altered in a program.
For example: 1, 2.5, "C programming is easy" etc.
As mentioned, an identifier also can be defined as a constant.

```
const double PI = 3.14
```

Here, *PI* is a constant. Basically what it means is that, *PI* and *3.14* is same for this program.

C language supports several types constants. Those are

1. Boolean Constants
2. Character Constants
3. Integer Constants
4. Real Constants
5. Complex Constants
6. String Constants

1. **Boolean Constants:** A Boolean data type can take only two values. Therefore, we expect that we have only two symbols to represent a Boolean type. The values are **true** or **false**. A Boolean value can have only two possible values: 0 (false) and 1 (true).

2. **Character Constants:** Character constants are enclosed between two single quotes(apostrophes)

   Example: 'a', 'A',' k'.

   Escape Sequences: Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. For example: newline (enter), tab, question mark etc. In order to use these characters, escape sequence is used.

   For example: `\n` is used for newline. The backslash (`\`) causes "escape" from the normal way the characters are interpreted by the compiler.

Escape Sequences	
Escape Sequences	**Character**
'\b'	Backspace
'\f'	Form feed
'\n'	Newline
'\r'	Return
'\t'	Horizontal tab
'\v'	Vertical tab
'\\'	Backslash

'\''	Single quotation mark
'\"'	Double quotation mark
'\?'	Question mark
'\0'	Null character

3. **Integer Constants:** An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:

- decimal constant(base 10)
- octal constant(base 8)
- hexadecimal constant(base 16)

For example:

Decimal constants: 0, -9, 22 etc
Octal constants: 021, 077, 033 etc
Hexadecimal constants: 0x7f, 0x2a, 0x521 etc

In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.

4. **Real Constants:** A floating point or real constant is a numeric constant that has either a fractional form or an exponent form. The default form for real constants is double. If we want resulting data type to be float or long double, we must use a code to specify the desired data type. As you might anticipate, f and F are used for float and l and L used for long double.

For example:

Representation	**Value**	**Type**
0.	0.0	double
.0	0.0	double
2.0	2.0	double
3.1416f	3.1416	float
3.1415926536L	3.1415926536	long double

5. **Complex Constants**: Complex constants are coded as two parts, the real part and the imaginary part, separated by a plus sign. The part is coded using the real format rules. The imaginary part is coded as a real number times (*) the imaginary constant (Complex I). The default form for complex constants is double.

   For Example:

Representation	Value	Type
12.3+14.4 * I	12.3 + 14.4 * $(-1)^{1/2}$	double complex
14F + 16F* I	14+ 16 * $(-1)^{1/2}$	float complex

6. **String Constants:** A string constant is a sequence of zero or more characters enclosed in double quotes.

`"good"`	`//string constant`
`""`	`//null string constant`
`"      "`	`//string constant of six white space`
`"x"`	`//string constant having single character.`
`"Earth is round\n"`	`//prints string with new line`

It is important to understand the difference between the null character and an empty string. The null character represents no value. As a character, it is 8 zero bits. An empty string is a string containing nothing.

'\0'	Null Character
" "	Empty String

Coding Constants: There are three different ways to code these constants in our program. Those are

1. Literal Constants
2. Defined Constants
3. Memory Constants

1. **Literal Constants:** A literal is an unnamed constant used to specify the data.
Ex: a=b+5; Here 5 is a literal constant.

2. **Defined Constants:** In this method in order to define a constant we use pre-processor command define. Like all pre-processor commands, it is prefaced with the pound sign (#).
Syntax: #define identifier value
Ex: #define pi 3.1416

3. Memory Constants: Memory constants use a C type qualifier, const, to indicate that the data cannot be changed. Its format is:
Syntax: const type identifier=value;
Ex: const float pi=3.14159;

Strings

➢ **Definition:** A string is a sequence of characters enclosed in a pair of double quotes(" ")

Ex: "Mahesh"

Special Symbols: The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

[] () {} , ; : * … = #

- **Braces {}:** These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.
- **Parentheses ():** These special symbols are used to indicate function calls and function parameters.
- **Brackets []:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

Operators: An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Special Operators

- **Arithmetic Operators:** An **arithmetic operator** is a **mathematical** function that takes two operands and performs a calculation on them. These are operators used to perform some basic arithmetic operations. Some of the Arithmetic operators listed below.

Operator	Description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denumerator
%	remainder of division
++	Increment operator increases integer value by one
--	Decrement operator decreases integer value by one

Examples: Assume A=10 and B=20 then

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2

%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 19

Sample program on Arithmetic operators:

```
#include <stdio.h>

main()
{
  int a = 21;
  int b = 10;
  int c ;
  c = a + b;
  printf("Line 1 - Value of c is %d\n", c );
  c = a - b;
  printf("Line 2 - Value of c is %d\n", c );
  c = a * b;
  printf("Line 3 - Value of c is %d\n", c );
  c = a / b;
  printf("Line 4 - Value of c is %d\n", c );
  c = a % b;
  printf("Line 5 - Value of c is %d\n", c );
  c = a++;
  printf("Line 6 - Value of c is %d\n", c );
  c = a--;
  printf("Line 7 - Value of c is %d\n", c );
}
```

OUTPUT:-

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

- **Relational operators:** A **relational operator** is a programming language construct or **operator** that tests or defines some kind of relation between two entities.

The following table shows all relation operators supported by C.

Operator	Description
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

For example let A=10 and B=20 then

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If values are not equal then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Sample program on Relational operators:

```
#include <stdio.h>
main()
{
            int a = 21;
            int b = 10;
            int c ;
              if( a == b )
            {
                printf("Line 1 - a is equal to b\n" );
             }
```

```
              else
               {
                  printf("Line 1 - a is not equal to b\n" );
                }

               if ( a < b )
              {
                   printf("Line 2 - a is less than b\n" );
               }
              else
              {
                   printf("Line 2 - a is not less than b\n" );
               }

               if ( a > b )
              {
                   printf("Line 3 - a is greater than b\n" );
               }
              else
              {
                      printf("Line 3 - a is not greater than b\n" );
                }

                /* Lets change value of a and b */
               a = 5;
              b = 20;
              if ( a <= b )
              {
                  printf("Line 4 - a is either less than or equal to  b\n" );
               }
               if ( b >= a )
              {
                 printf("Line 5 - b is either greater than  or equal to b\n" );
              }
}
```

OUTPUT:-

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to
```

- **Logical Operators:** Logical operators are used to combine two or more conditions for comparision.

  C language supports following 3 logical operators. Suppose a=1 and b=0,

Operator	Description	Example

&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Sample program on Logical operators:

```
#include <stdio.h>
main()
{
          int a = 5;
          int b = 20;
         int c ;
         if ( a && b )
        {
            printf("Line 1 - Condition is true\n" );
         }

         if ( a || b )
        {
           printf("Line 2 - Condition is true\n" );
        }

        /* lets change the value of  a and b */
          a = 0;
          b = 10;

          if ( a && b )
          {
              printf("Line 3 - Condition is true\n" );
           }
        else
        {
            printf("Line 3 - Condition is not true\n" );
         }

         if ( !(a && b) )
        {
              printf("Line 4 - Condition is true\n" );
         }

}
```

OUTPUT:

```
Line 1 - Condition is true
```

Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- **Bitwise Operators:** Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to **float** or **double**. C supports several Bitwise Operators. Those are listed below

Assume A=60 and B=13

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e,. 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1

The truth table for bitwise &, | and ^

The bitwise shift operators shift the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value are to be shifted. Both operands have the same precedence.

Example :

```
a = 0001000
b= 2
a << b = 0100000
a >> b = 0000010
```

Sample program on Bitwise operators:

```
#include <stdio.h>
main()
{
   unsigned int a = 60;           /* 60 = 0011 1100 */
   unsigned int b = 13;           /* 13 = 0000 1101 */
   int c = 0;
   c = a & b;       /* 12 = 0000 1100 */
   printf("Line 1 - Value of c is %d\n", c );
   c = a | b;       /* 61 = 0011 1101 */
   printf("Line 2 - Value of c is %d\n", c );
   c = a ^ b;       /* 49 = 0011 0001 */
   printf("Line 3 - Value of c is %d\n", c );
   c = ~a;          /*-61 = 1100 0011 */
   printf("Line 4 - Value of c is %d\n", c );
   c = a << 2;     /* 240 = 1111 0000 */
   printf("Line 5 - Value of c is %d\n", c );
   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Line 6 - Value of c is %d\n", c );
}
```

OUTPUT:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
```

- **Assignment Operators:** These operators are used to assign the values to the variable.

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A

/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Sample program on Assignment operators:

```
#include <stdio.h>
main()
{
  int a = 21;
  int c ;
  c =  a;
  printf("Line 1 - =  Operator Example, Value of c = %d\n", c );
  c +=  a;
  printf("Line 2 - += Operator Example, Value of c = %d\n", c );
  c -=  a;
  printf("Line 3 - -= Operator Example, Value of c = %d\n", c );
  c *=  a;
  printf("Line 4 - *= Operator Example, Value of c = %d\n", c );
  c /=  a;
  printf("Line 5 - /= Operator Example, Value of c = %d\n", c );
  c  = 200;
  c %=  a;
  printf("Line 6 - %= Operator Example, Value of c = %d\n", c );
  c <<=  2;
  printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );

  c >>=  2;
  printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );
  c &=  2;
  printf("Line 9 - &= Operator Example, Value of c = %d\n", c );
  c ^=  2;
  printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );
  c |=  2;
  printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
}
```

OUTPUT:

```
Line 1 - = Operator Example, Value of c = 21
```

Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
Line 8 - >>= Operator Example, Value of c = 11
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2

- **Conditional operator*:*** It is also known as ternary operator and used to evaluate conditional expression.

  ```
  Syntax: epr1? expr2 : expr3
  ```

  If **epr1** Condition is true ? Then value **expr2** : Otherwise value **expr3**

- **Special Operators:**

- **Increment and Decrement Operators:**
  - C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.
  - Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1.

Operator	Description	Example
sizeof	Returns the size of an variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of an variable	**&x ;** return address of the variable **x**
*	Pointer to a variable	***x ;** will be pointer to a variable **x**

These two operators are unary operators, meaning they only operate on a single operand.

Sample program on Assignment operators:

/ C Program to demonstrate the working of increment and decrement operators

```
#include <stdio.h>
int main()
```

```
{
     int a = 10, b = 100;
     float c = 10.5, d = 100.5;
     printf("++a = %d \n", ++a);
     printf("--b = %d \n", --b);
     printf("++c = %f \n", ++c);
     printf("--d = %f \n", --d);
     return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Expressions:

Conditional Statements: Conditional statements are used to execute a statement or a group of statement based on certain conditions. C language supports several Conditional Statements. Those are

1. if
2. if else
3. else if
4. switch
5. goto

1) if conditional statement in C : If the condition is true the statements inside the parenthesis { }, will be executed, else the control will be transferred to the next statement after it.

Syntax for if statement in C:

```
if(condition)
{
Valid C Statements;
}
```

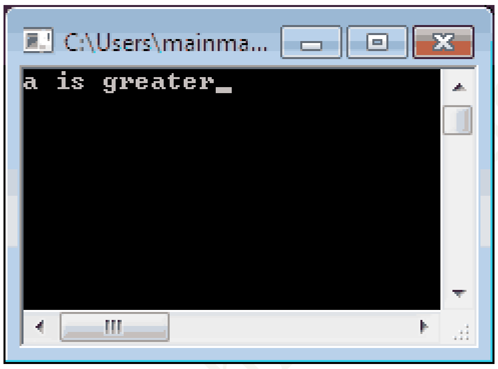
If the condition is true the statements inside the parenthesis { }, will be executed, else the control will be transferred to the next statement after it.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
a=10;
b=5;
if(a>b)
{
printf("a is greater");
}
getch();
}
```

Program Algorithm / Explanation:

1. **#include<stdio.h>** header file is included because; the C in-built statement **printf** we used in this program comes under stdio.h header files.
2. **#include<conio.h>** is used because the C in-built function **getch()** comes under conio.h header files.
3. **main ()** function is the place where C program execution begins.
4. Two variables **a&b** of type **int** is declared.
5. Variable **a** is assigned a value of **10** and variable **b** with **5**.
6. A **if** condition is used to check whether **a** is greater than **b**. **if(a>b)**
7. As **a** is greater than **b** the printf inside the **if { }** is executed with a message "**a is greater**".

Output:



```
a is greater_
```

Flow-chart for simple if statement

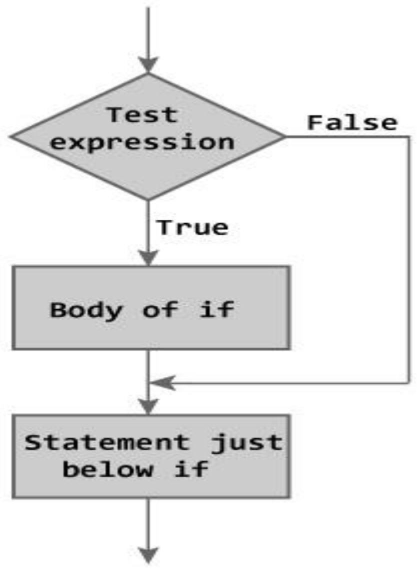


Figure: Flowchart of if Statement

2) if else in C : In if else if the condition is true the statements between if and else is executed. If it is false the statement after else is executed.

Syntax for if - else:

```
if(condition)
{
Valid C Statements;
}
else
{
Valid C Statements;
}
```

In if else if the condition is true the statements between if and else is executed. If it is false the statement after else is executed.

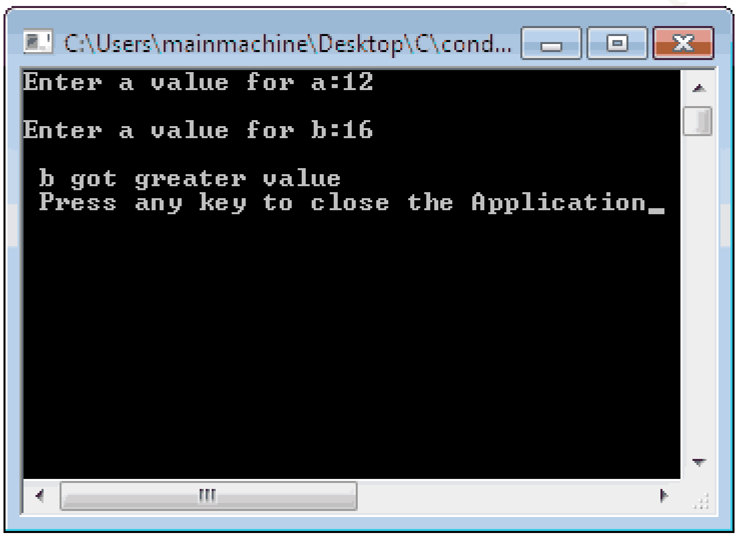
Sample Program :

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
printf("Enter a value for a:");
scanf("%d",&a);
printf("\nEnter a value for b:");
scanf("%d",&b);

if(a>b)
{
printf("\n a got greater value");
}
else
{
printf("\n b got greater value");
}
printf("\n Press any key to close the Application");
getch();
}
```

Program Algorithm / Explanation :

1. **#include<stdio.h>** header file is included because; the C in-built statement **printf** we used in this program comes under stdio.h header files.
2. **#include<conio.h>** is used because the C in-built function **getch()** comes under conio.h header files.
3. **main ()** function is the place where C program execution begins.
4. Two integer variable **a and b** are declared.
5. User Input values are received for both **a and b** using **scanf**.
6. An **if else** conditional statement is used to check whether **a** is greater than **b**.
7. If **a** is greater than **b**, the message "**a got greater value**" is displayed using **printf**.
8. If **b** is greater the message "**b got greater value**" is displayed.

Output :



```
Enter a value for a:12

Enter a value for b:16

 b got greater value
 Press any key to close the Application_
```

Flow-chart for simple if – else statement

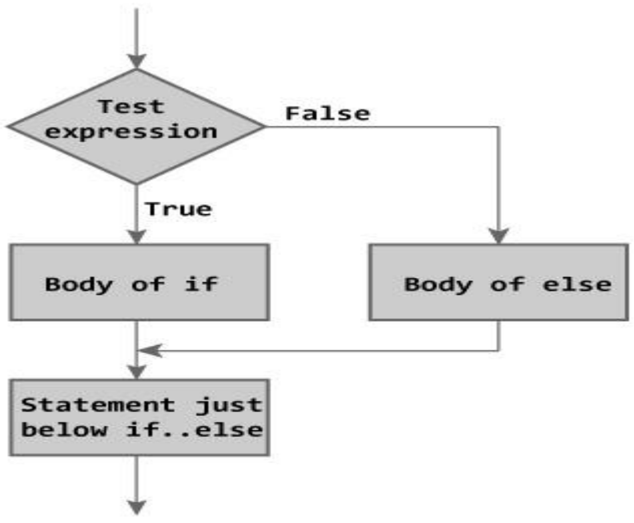
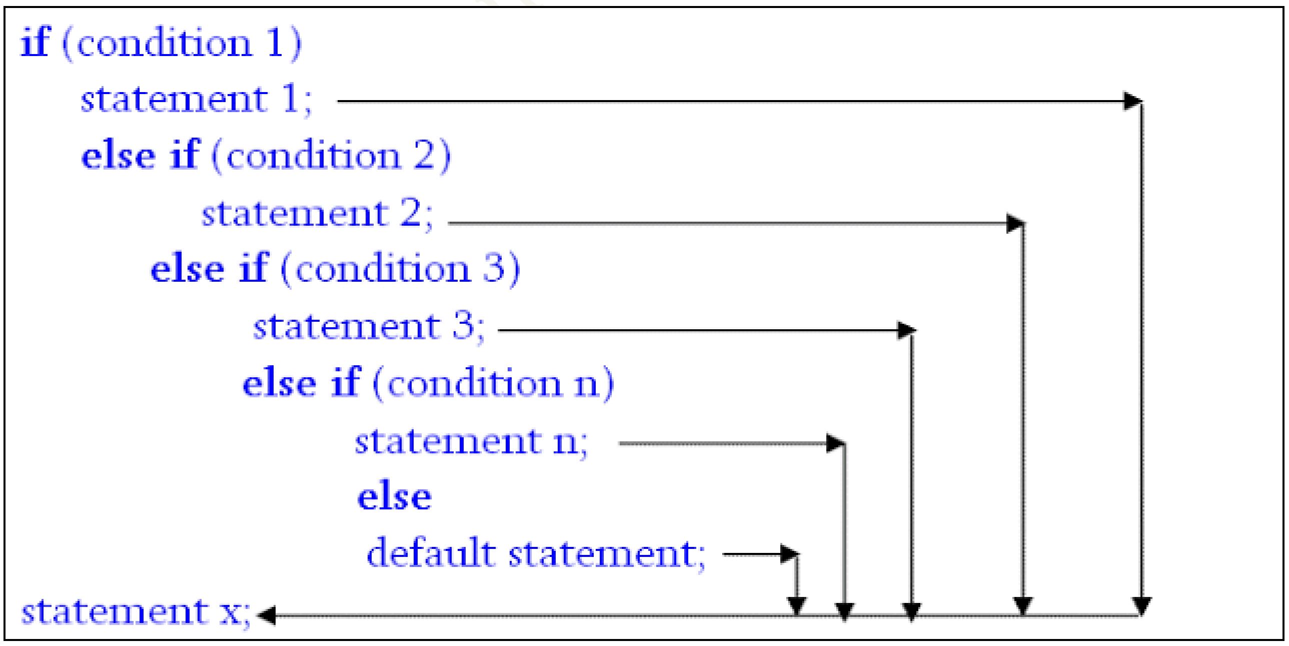


Figure: Flowchart of if...else Statement

3) else-if ladder in C: In else if, if the condition is true the statements between if and else if is executed. If it is false the condition in else if is checked and if it is true it will be executed. If none of the condition is true the statement under else is executed.

Syntax :

In else if, if the condition is true the statements between if and else if is executed. If it is false the condition in else if is checked and if it is true it will be executed. If none of the condition is true the statement under else is executed.



```
if (condition 1)
    statement 1;
    else if (condition 2)
            statement 2;
        else if (condition 3)
              statement 3;
              else if (condition n)
                    statement n;
                    else
                     default statement;
statement x;
```

Sample program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
printf("Enter a value for a:");
scanf("%d",&a);
printf("\nEnter a value for b:");
scanf("%d",&b);
if(a>b)
{
printf("\n a is greater than b");
}
else if(b>a)
{
printf("\n b is greater than a");
}
else
{
printf("\n Both a and b are equal");
}
printf("\n Press any key to close the application");
getch();
}
```

Program Algorithm / Explanation :

1. **#include<stdio.h>** header file is included because; the C in-built statement **printf** we used in this program comes under stdio.h header files.
2. **#include<conio.h>** is used because the C in-built function **getch** comes under conio.h header files.
3. **main ()** function is the place where C program execution begins.
4. Two variables **a & b** of type **int** are declared.
5. Values for a & b are received from user through **scanf**.
6. **if** condition is used to check whether **a** is greater than **b**. **if(a>b)**

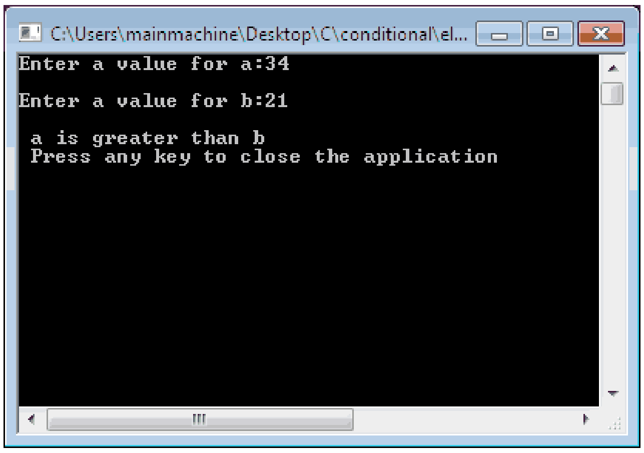
If the condition is true the statement between if and else if is executed.

1. If the condition is not satisfied **else if()** condition is checked. **Else if(b>a)**

If **b** is greater than **a** the statement between **else if** and **else** is executed.

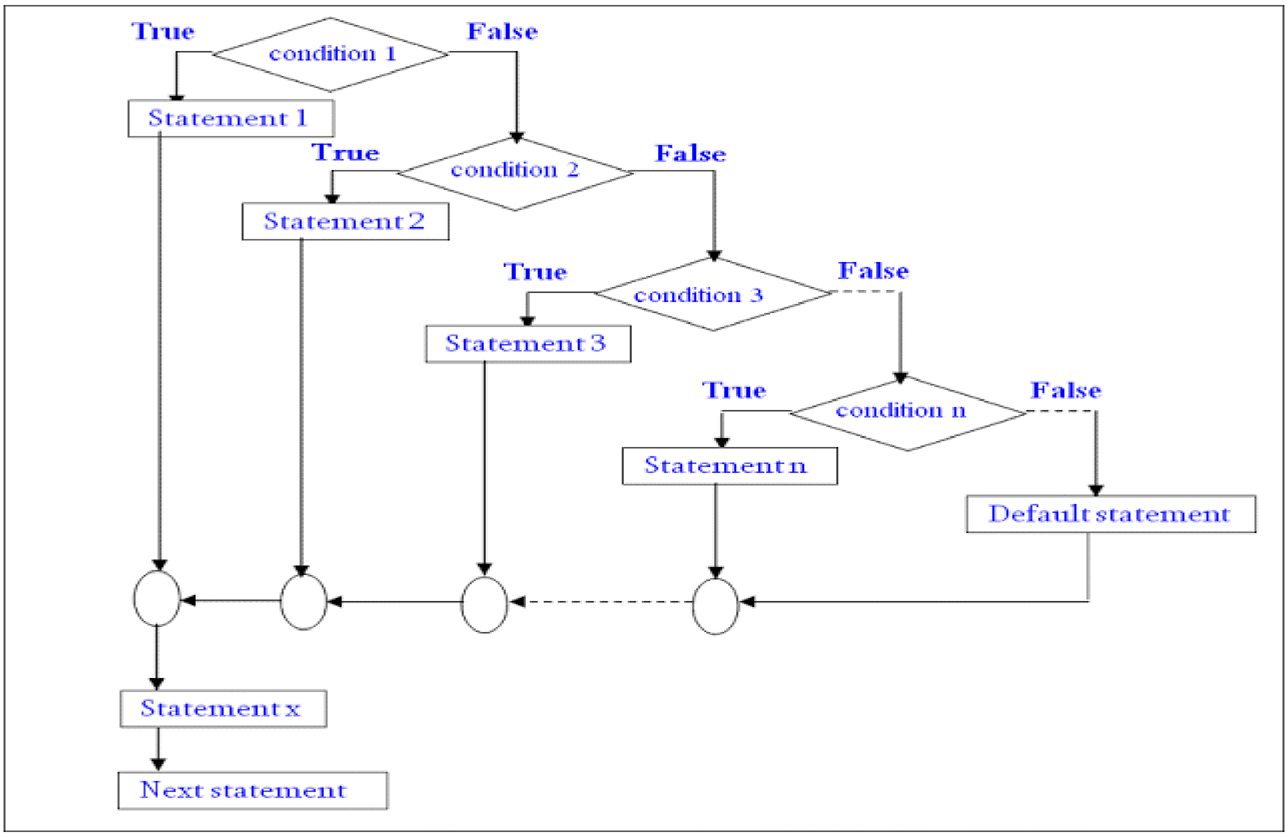
1. If both the conditions are **false** the statement after **else** is executed.

Output :



```
Enter a value for a:34

Enter a value for b:21

 a is greater than b
 Press any key to close the application
```

Flow-chart for else-if ladder



4) Switch statement in C: Switch statements can also be called matching case statements. If matches the value in variable (switch (variable)) with any of the case inside, the statements under the case that matches will be executed. If none of the case is matched the statement under default will be executed.

Syntax :

```
switch(variable)
{
case 1:
```

```
Valid C Statements;
break;
-
-
case n:
Valid C Statements;
break;
default:
Valid C Statements;
break;
}
```

Switch statements can also be called matching case statements. If matches the value in variable (switch (variable)) with any of the case inside, the statements under the case that matches will be executed. If none of the case is matched the statement under default will be executed.

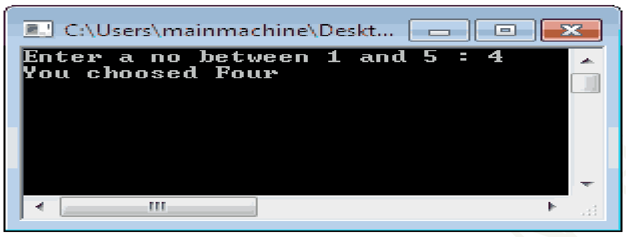
Sample program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a;
printf("Enter a no between 1 and 5 : ");
scanf("%d",&a);

switch(a)
{
case 1:
printf("You choosed One");
break;
case 2:
printf("You choosed Two");
break;
case 3:
printf("You choosed Three");
break;
case 4:
printf("You choosed Four");
break;
case 5:
printf("You choosed Five");
break;
default :
printf("Wrong Choice. Enter a no between 1 and 5");
break;
}
getch();
}
```

Program Algorithm / Explanation :

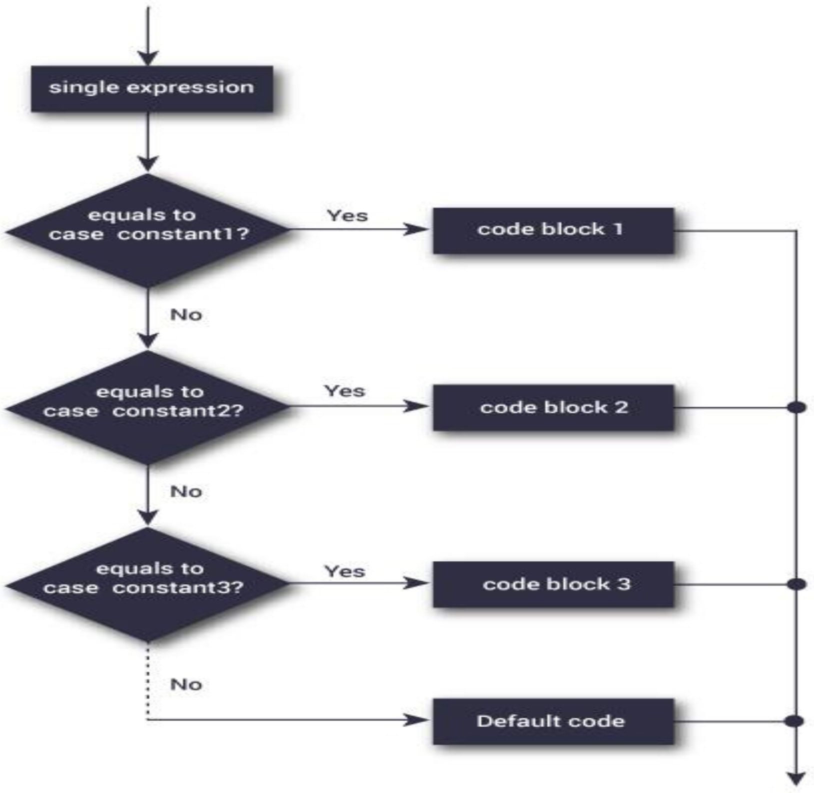
1. **#include<stdio.h>** header file is included because; the C in-built statement **printf** we used in this program comes under stdio.h header files.
2. **#include<conio.h>** is used because the C in-built function **getch()** comes under conio.h header files.
3. **main ()** function is the place where C program execution begins.
4. Variable **a** of type **int** is declared.
5. **scanf** is used to get the value from user for variable **a**.
6. variable **a** is included for **switch case**. **switch(a)**.
7. If the user Input was **1** the statement inside **case 1:** will be executed. Likewise for the rest of the case's until **5**.
8. But if the user Input is other than **1 to 5** the statement under **default :** will be executed.

Output :



```
Enter a no between 1 and 5 : 4
You choosed Four
```

Flow-chart for switch case



5) goto statement in C :goto is a unconditional jump statement.

Syntax:

```
goto label;
```

So we have to use the goto carefully inside a conditional statement.

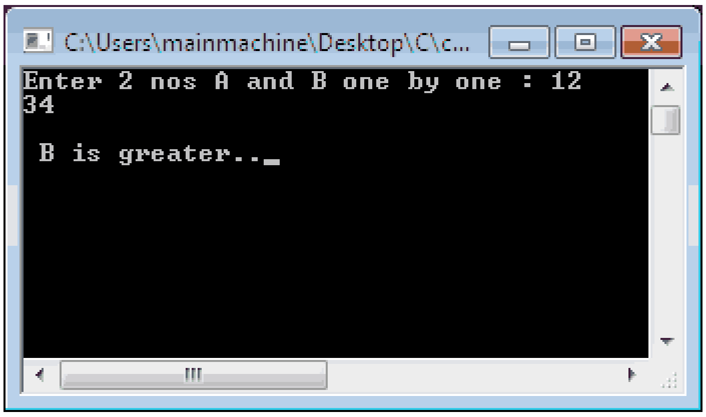
Sample program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
printf("Enter 2 nos A and B one by one : ");
scanf("%d%d",&a,&b);
if(a>b)
{
goto first;
}
else
{
goto second;
}

first:
printf("\n A is greater..");
goto g;

second:
printf("\n B is greater..");

g:
getch();
}
```

Program Algorithm / Explanation:

1. **#include<stdio.h>** header file is included because; the C in-built statement **printf** we used in this program comes under stdio.h header files.
2. **#include<conio.h>** is used because the C in-built function **getch()** comes under conio.h header files.
3. **main ()** function is the place where C program execution begins.
4. Two variables **a & b** of type **int** are declared.
5. User Inputs are received for **a & b** using **scanf**.
6. **if** condition is used to check whether **a** is greater than **b**. **if(a>b)**.
7. if it is true **goto** statement is used to jump to the label **first :** and the statement under **first :** is executed and then again jump is performed to get to the end of the program. **goto g**:
8. if the condition is **false** a statement is used to jump to label **second**. **goto second :** and the statement under second : is executed.

Output:

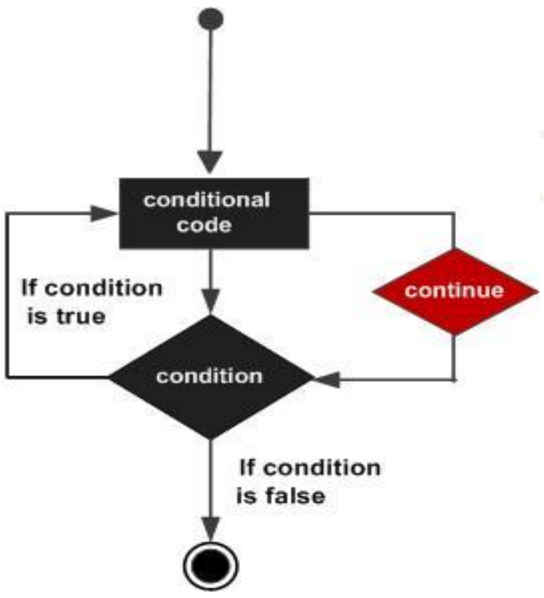


```
Enter 2 nos A and B one by one : 12
34

 B is greater..
```

Continue Statement in C: Continue statement is mostly used inside loops. Whenever it is encountered inside a loop, control directly jumps to the beginning of the loop for next iteration, skipping the execution of **statements** inside loop's body for the current iteration.

Syntax: `continue;`

Flow Diagram



Example:

```
#include <stdio.h>
 int main()
{
  /* local variable definition */
  int a = 10;
  /* do loop execution */
  do
  {
          if( a == 15)
           {
              /* skip the iteration */
                a = a + 1;
                continue;
            }

            printf ("value of a: %d\n", a);
            a++;

    } while ( a < 20 );
    return 0;
}
```

OUTPUT:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Iterative Statements/ Loops: Loops are used in programming to repeat a specific block until some end condition is met. There are three loops in C programming:

1. **for loop**
2. **while loop**
3. **do...while loop**

for loop:

The syntax of a for loop is:

```
for (initialization Statement; test Expression; update Statement)
{
     // block of statements
}
```

<u>How for loop works?</u>

The initialization statement is executed only once.

Then, the test expression is evaluated. If the test expression is false (0), for loop is terminated. But if the test expression is true (nonzero), codes inside the body of for loop is executed and update expression is updated. This process repeats until the test expression is false.

The *for* loop is commonly used when the number of iterations is known.

for loop Flowchart

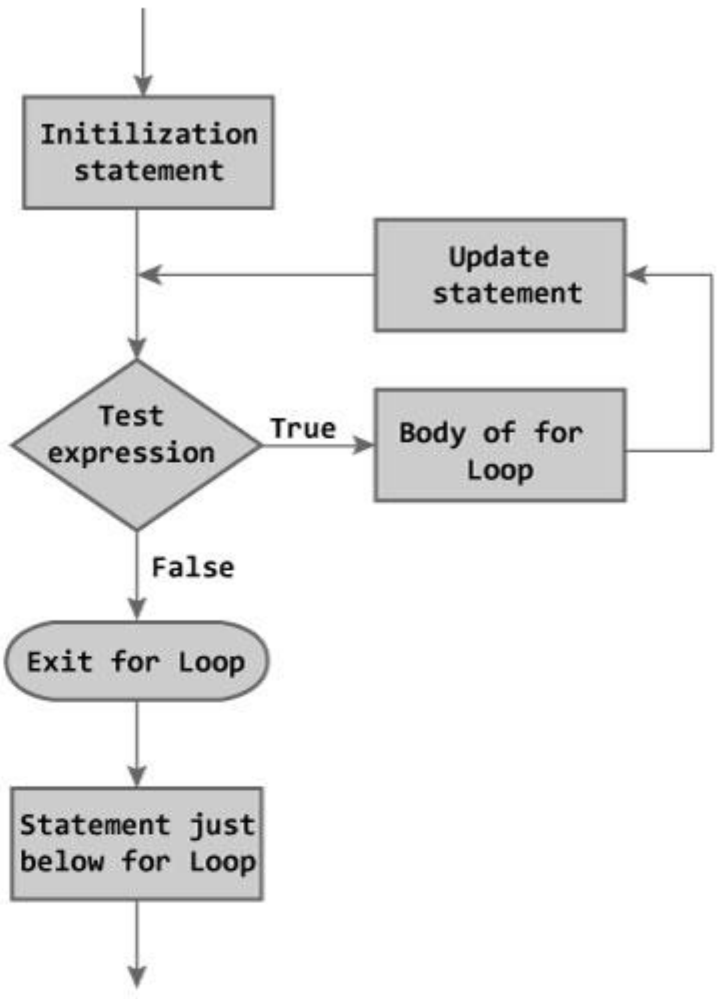


Figure: Flowchart of for Loop

Example: for loop

```
// Program to calculate the sum of first n natural numbers
// Positive integers 1,2,3...n are known as natural numbers

#include <stdio.h>
int main()
{
    int n, count, sum = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    // for loop terminates when n is less than count
    for(count = 1; count <= n; ++count)
    {
        sum += count;
    }
    printf("Sum = %d", sum);
    return 0;
}
```

Output:

```
Enter a positive integer: 10
```

```
Sum = 55
```

The value entered by the user is stored in variable *n*. suppose the user entered 10.

The *count* is initialized to 1 and the test expression is evaluated. Since, the test expression `count <= n` (1 less than or equal to 10) is true, the body of for loop is executed and the value of *sum* will be equal to 1.

Then, the update statement `++count` is executed and count will be equal to 2. Again, the test expression is evaluated. The test expression is evaluated to true and the body of for loop is executed and the *sum* will be equal to 3. And, this process goes on.

Eventually, the count is increased to 11. When the count is 11, the test expression is evaluated to 0 (false) and the loop terminates.

while loop:

The syntax of a while loop is:

```
while (testExpression)
{
   //codes
}
```

How while loop works?

The while loop evaluates the test expression.

If the test expression is true (nonzero), codes inside the body of while loop is evaluated. Then, again the test expression is evaluated. The process goes on until the test expression is false.

When the test expression is false, the while loop is terminated.

Flowchart of while loop

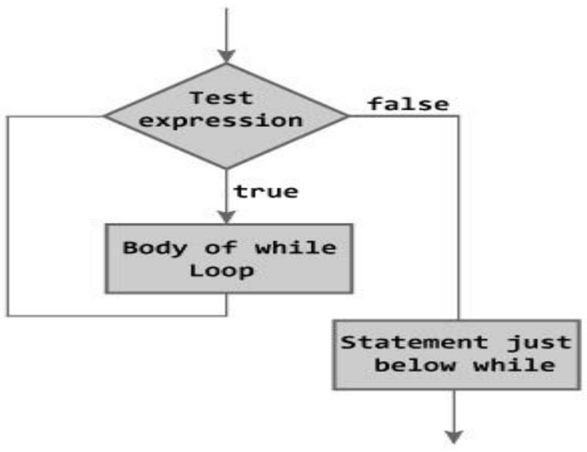


Figure: Flowchart of while Loop

Example #1: while loop

```
// Program to find factorial of a number
// For a positive integer n, factorial = 1*2*3...n

#include <stdio.h>
int main()
{
    int number;
    long long factorial;

    printf("Enter an integer: ");
    scanf("%d",&number);

    factorial = 1;

    // loop terminates when number is less than or equal to 0
    while (number > 0)
    {
        factorial *= number;  // factorial = factorial*number;
        --number;
    }

    printf("Factorial= %lld", factorial);

    return 0;
}
```

Output

```
Enter an integer: 5
Factorial = 120
```

do...while loop:

The `do..while` loop is similar to the `while` loop with one important difference. The body of `do...while` loop is executed once, before checking the test expression. Hence, the `do...while loop` is executed at least once.

do...while loop Syntax

```
do
{
  // codes
}
while (test Expression);
```

How do...while loop works?

The code block (loop body) inside the braces is executed once.

Then, the test expression is evaluated. If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to 0 (false). When the test expression is false (nonzero), the `do...while` loop is terminated.

Flow- Control of do--while

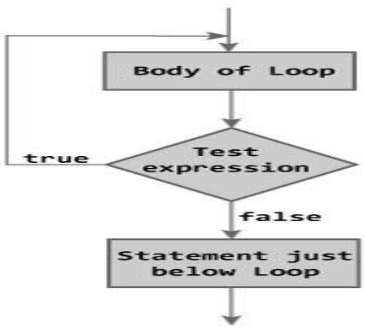


Figure: Flowchart of do...while Loop

Example: **do...while loop**

```
// Program to add numbers until user
enters zero

#include <stdio.h>
int main()
{
        double number, sum = 0;

        // loop body is executed at least once
         do
         {
                printf("Enter a number: ");
                scanf("%lf", &number);
                sum += number;
         }while(number != 0.0);

    printf("Sum = %.2lf",sum);

    return 0;
}
```

Output

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

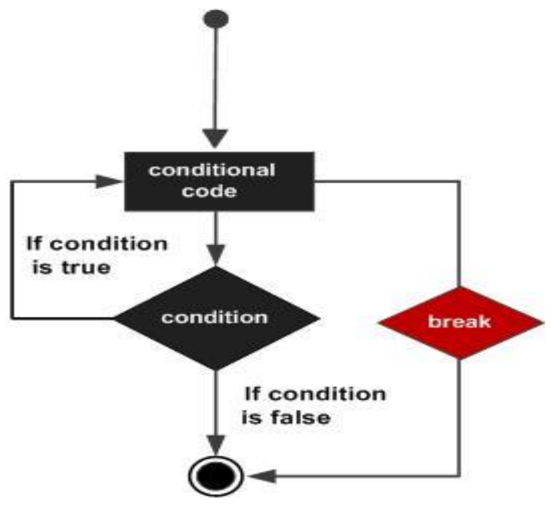
Break statement in C: The **break** statement in C programming has the following two usages −

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax: break;

Flow Diagram



Example

```
#include <stdio.h>

int main ()
{
        /* local variable definition */
        int a = 10;
        /* while loop execution */
        while( a < 20 )
        {
              printf("value of a: %d\n", a);
              a++;
              if( a > 15)
              {
                /* terminate the loop using break statement */
                break;
              }

         }

   return 0;
}
```

OUTPUT:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```